INF-3990

MASTER'S THESIS IN

INFORMATICS

Improving the performance of the in-band byte stream file transfer XMPP extension

Karsten Orlin

24.09.2009

FACULTY OF SCIENCE
Department of Computer Science
University of Tromsø

Preface

My work on this thesis is coming to an end. This thesis was presented to me by Johan Gustav Bellika as a possible assignment for a master student. The technology in question was entirely new to me (except for Java), although the concept of instant messaging and file transfer was a familiar one. Even though I did not reach the goals I set out to accomplish, it has been an interesting learning experience for me and given me much insight into XMPP in general and the Openfire XMPP server specifically.

Acknowledgement

There are people that I would like to thank for their support during the work on my thesis.

First and foremost I would like to thank my supervisor, Johan Gustav Bellika for helping me in my work on this thesis and for his availability to me. It has been good to know that he always could be reached, when needed. Thank you! I would also like to thank Lars Ilebrekke and Taridzo Chomutare at Norwegian Centre for Telemedicine (NST) for helping me with many of my Java, Eclipse and Openfire related issues. Kristian Andreassen at has also been helpful in answering my XMPP and Openfire questions and I'm also thankful for his willingness to help me set up a test environment within the Norwegian Health Network, even though I never got to that stage.

Last but not least, I would like to thank my friend and savior, Jesus, for giving me the faith and the strength to not give up and quit when it seemed like I would never be able to finish my thesis on time.

Summary

In looking at the XMPP protocol as an alternative to the ordinary way of transferring files within a health network setting, namely e-mail, performance and security are important factors to consider. For security reasons we preferred to use in-band over out-of-band file transfer. The tradeoff is that this method puts a higher strain on the XMPP server and is significantly slower than its counterpart, out-of-band. In researching a specific XMPP implementation, the Openfire XMPP server, and looking into how it deals with in-band file transfers, we have found some ways to increase in-band file transfer performance, but not in the originally intended way, which would be through improvements in the Openfire source code concerning in-band file transfers.

Contents

Figure list

Table list

Code listing

1 Introduction

This chapter outlines the background and defines the primary goal of this thesis.

1.1 Background

The Norwegian Health Network (NHN) is used for secure electronic exchange of various kinds of patient health care data, between hospitals, from hospitals to general practitioners and to pharmacists. These data may be discharge letters, lab results, referrals and radiology results and are mainly communicated through the SMTP/POP protocols.

The Extensible Messaging and Presence Protocol (XMPP) protocol was originally developed for instant messaging (IM) and presence services. However, the protocol has been enhanced with numerous XMPP Extension Protocols (XEP's). Two of these extensions, XEP-0047 In-band bytestream (XEP-0047) and XEP-0065 Socks 5 bytestream (XEP-0065) are designed specifically for file transfer. The reliability of the XMPP protocol for file transfer within NHN was examined by Andreassen as a potential alternative to the SMTP/POP protocol. It was observed that both XEP's were working reliably with the XMPP core protocol [1]. XEP-0047 did perform poorly compared to the XEP-0065 extension, but has the advantage of not being dependant on a proxy server outside the firewalled/NAT environment, thereby avoiding having to open an additional port. This is advantageous in our health network setting, as opening more ports requires a more comprehensive risk and security analysis in addition to a higher demand for resources in the operating department. This is why we will seek ways to improve file transfer performance using in-band bytestream (IBB).

1.2 Problem definition

The goal of this thesis is to improve the performance of the in-band byte stream file transfer XMPP extension.

1.3 Method

There are three major paradigms in computer science [2] . The first paradigm, theory, is rooted in mathematics and consists of four steps followed in the development of a coherent, valid theory:

- Characterize objects of study (definition).
- Hypothesize possible relationships among them (theorem).
- Determine whether the relationships are true (proof).
- Interpret results.

The second paradigm, abstraction (modeling), is rooted in the experimental scientific method and consists of four stages that are followed in the investigation of a phenomenon:

- Form a hypothesis.
- Construct a model and make a prediction.
- Design an experiment and collect data.
- Analyze results.

The third paradigm, design, is rooted in engineering and consists of four steps followed in the construction of a system (or device) to solve a given problem:

- State requirements.
- State specifications.
- Design and implement the system.
- Test the system.

Our main focus is not to design and implement a new system, but rather to tweak or add to an existing one to improve it in the area of file transfer performance. Most of our work will be focused around a cycle of tweaking, testing and analyzing results, which means that we will work within the boundaries of both the abstraction and the design paradigm.

A given implementation can seek performance or improvement and enhancement of prior implementations (proof of performance), demonstrate that a particular configuration of ideas or an approach achieves its objectives (proof of concept), or demonstrate a fundamentally new computing phenomenon (proof of existence) [2] . We will seek to improve and enhance prior implementations, which mean that we belong in the proof of performance category.

We will attempt to optimize a XMPP server's file transfer performance. For this we have chosen Jive Software's Openfire XMPP server [3]. One reason for selecting this XMPP server implementation over others like Tigase [4] and ejabberd [5] is that we already are familiar with the Java programming language, which is Openfire's programming language. Another reason is the support available to us from some of my supervisor's coworkers at NST, as Openfire is at the base of some of the company's work, and that brings us to a third motivation, as any positive result might be relevant to current NST projects involving Openfire. Subjectively speaking, it also seems to be the most polished and professional implementation around, although that's merely the author's impression and not based on close scrutiny of all the alternatives.

We have tried to understand the Openfire code by digging into the source material, finding critical spots for file transfers, debugging them and attempting to understand which segment of code did what. Technically, this involved installing the Eclipse integrated development environment (IDE) [6] on a computer with Windows XP, the Openfire source code and setting up two XMPP Spark clients from Jive Software to talk to each other through a Windows 2000 virtual machine on the same computer, to simulate two computers' client communication. This setup, displayed in Figure 1, was obviously not intended for performance testing, but to act as a simple development and debugging platform.

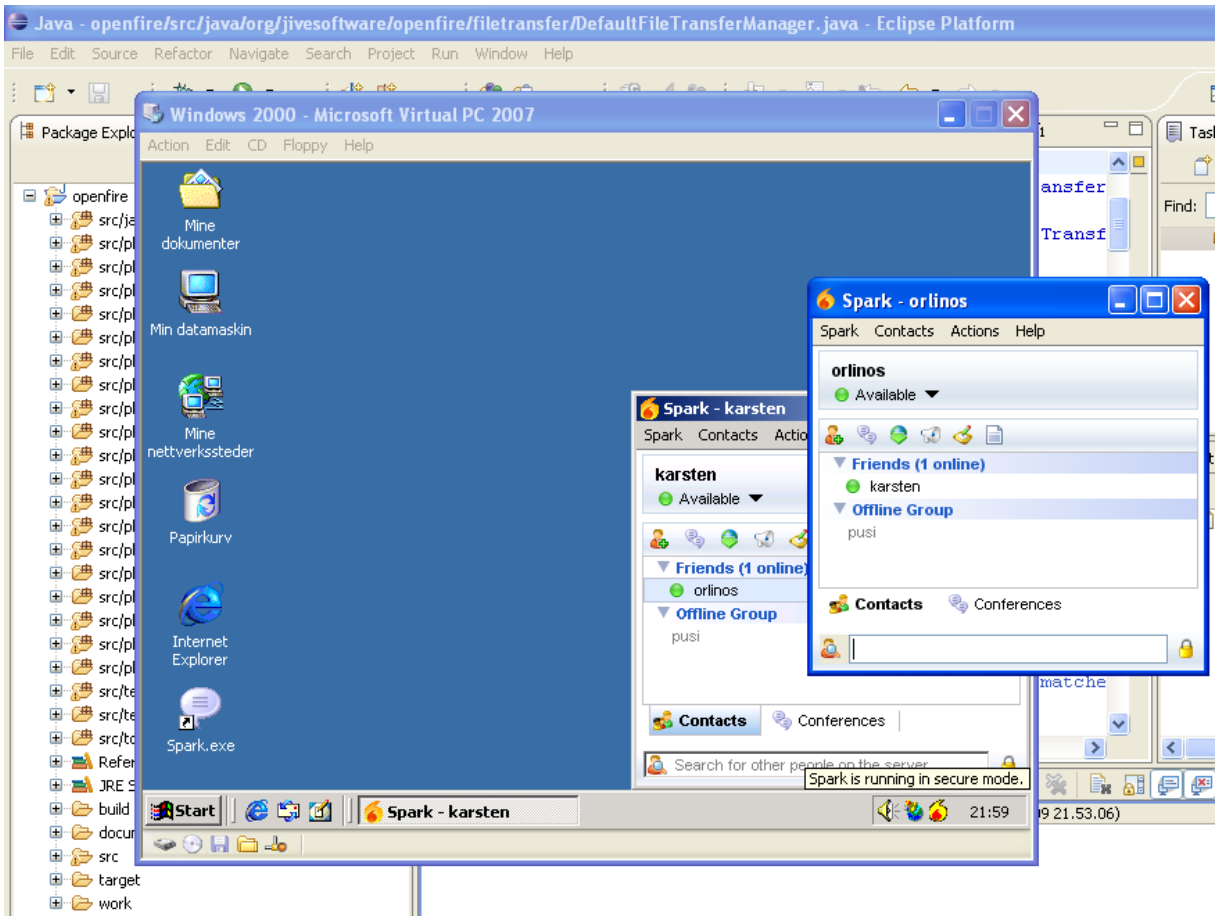

Figure 1: Our development environment

The experiment will be considered a success if we find a way to improve the Openfire source code in the area of in-band file transfer and can prove an increased performance when we test it within the Norwegian Health Network, under various loads and file sizes.

1.4 Scope and limitations

Our goal is to improve in-band file transfer performance, but we will limit ourselves to what can be tweaked and changed in the Openfire server code. As a consequence, there is also a limitation concerning standard Java libraries that the server uses. We will not try to replace those with better or more efficient ones.

1.5 Main results

The result of this thesis is that we could not find a way to optimize the source code for file transfer. Our time was spent trying to understand the chosen XMPP server's complex java code and structure, and the search for ways to optimize was fruitless. There did not present itself any apparent way to optimize the code in this area. However, we present some alternate steps that would increase in-band file transfer performance.

1.6 Summary

This chapter has given an overview of this thesis, the background and problem definition, description of methods used, scope and limitations and a presentation of the main results..

2 Theory and background

This chapter will give some background about the technologies and tools discussed and used in this thesis work. We will look at the core Jabber protocol in some detail, examine the relevant XMPP extensions for this thesis, and then explain our testing environment, the Norwegian Health Network. Furthermore, we will look at our XMPP server of choice, Jive Software's Openfire XMPP server.

2.1 Jabber/XMPP

This section gives a look into the various aspects of XMPP. The terms XMPP and Jabber will be used interchangingly, as they refer to the same thing.

2.1.1 Jabber history

The following is the first mention of Jabber/XMPP to surface on the Internet and was posted on the website www.slashdot.org at the beginning of 1999 [7]. It included a statement from the developer Jeremie Miller about the new project he was working on:

> *"Jabber is a new project I recently started to create a complete open-source platform for Instant Messaging with transparent communication to other IM systems (ICQ, AIM, etc). Most of the initial design and protocol work is done, as well as a working server and a few test clients."*

Soon, a core group of developers joined Miller to build out the server as well as open-source Jabber clients for Windows and Linux, and various other components [8]. This group also defined an open wire protocol for XML streaming, which is now an important part of the XMPP protocol. In 2001, Jabber Software Foundation (JSF) was formed to function as a standards development organization for the Jabber community [8]. In 2002, JSF submitted the XML streaming protocols to the Internet Engineering Task Force (IETF) as XMPP, and IETF approved it in 2004 and published four Requests for Comments (RFC) (RFC3920-23) [9]. By that, Jabber was formalized as XMPP. In 2006, JSF renamed its "Jabber Enhancement Proposals" (JEP) specification series to "XMPP Extension Protocols" (XEP). In 2007, the foundation itself was renamed from JSF to the XMPP Standards Foundation (XSF).

2.1.2 XMPP technology

XMPP is short for Extensible Messaging and Presence Protocol and that is just what XMPP is. It is extensible and it is a protocol for communicating instant messages between users, just like MSN Messenger, Yahoo Messenger and the like. The presence part of the name is about users

broadcasting their status to their friends. What makes XMPP different from most of its peers is that the protocol is open-source and anyone can develop and publish an implementation based on it.

XMPP is an Extensible Markup Language (XML) based protocol intended for use with IM. However, it has been extended for use in other areas, like network-management systems, online gaming networks, applications for financial trading, content syndication, and remote instrument monitoring [8]. The core XMPP protocol defines the core functionality of how clients and servers work together, and XMPP extensions (XEP) define functionality which goes beyond the basics, like mentioned above.

The XMPP protocol deals with streams and stanzas. In our setting, streams are like open channels established in both direction between the server and clients. The client queries the server and opens a stream, and then the server opens another stream towards the client (response stream). As mentioned, XML is the language by which the entities communicate. When the server recognizes a <stream> tag, it knows this is the beginning of a stream of data from the client. The </stream> tag would mark the end of communication. Between the beginning and end of a stream, stanzas are sent. The XMPP Core documentation [10] defines an XML stream as a container for the exchange of XML elements between any two entities over a network. It goes on to say:

> *The start of an XML stream is denoted unambiguously by an opening XML <stream> tag, while the end of the XML stream is denoted unambiguously by a closing XML </stream> tag. During the life of the stream, the entity that initiated it can send an unbounded number of XML elements over the stream, either elements used to negotiate the stream (e.g., to negotiate Use of TLS (Use of TLS) or Use of SASL (Use of SASL)) or XML stanza , see section 4.1 in* [10]

You could think of an XML stream as an envelope for the different XML stanzas sent while the stream is open as illustrated in figure 1. When the stream is closed, the underlying UDP or TCP connection (usually TCP) is also closed.

An XMP stanza is according to [10] a discrete semantic unit of structured information that is sent from one entity to another over an XML stream. It is a child of the root level, which is the mentioned <stream>. Stanzas come as <message/>, <presence/> and <iq/> elements and these contain data corresponding to their semantic value. In addition these stanzas have some common attributes. These are:

- to: JID of recipient
- from: JID of sender
- id: can be a unique ID assigned to each stanza
- type: varies by stanza
- xml:lang: used to specify human language

Table 1: Structure and contents of a XMPP stream session with stanzas, from section 4.1 in [10].

```
|--------------------|
| <stream>           |
|--------------------|
| <presence>         |
|   <show/>          |
| </presence>        |
|--------------------|
| <message to='foo'> |
|   <body/>          |
| </message>         |
|--------------------|
| <iq to='bar'>      |
|   <query/>         |
| </iq>              |
|--------------------|
| ...                |
|--------------------|
| </stream>          |
|--------------------|
```

The <presence /> stanza is a basic broadcast, or publish-subscribe mechanism. It is about users, or clients, passing their availability status to the other users in their friends list, or roster, as it is called. If you have ever used an IM service, you are probable aware of that you can see if your friends are online and whether they are busy or available to talk, or if they are away from the computer at the moment. An online XMPP client broadcasts (and receives) such presence data. All the users that have subscribed to you and are online will receive your online status. Technically, the client sends the data to a server. The server receives it and based on the particular user's friends list, or roster, it passes the presence data on to those users, who does the same to his or her online friends.

<message/> holds the message the user passes on to another user and <iq/> is a request-response mechanism for where a more structured data flow is required.

It is possible (and quite simple) to install and set up a private and isolated XMPP server on a corporate network. It gives the benefit of not having to deal with spam of any kind, since the server is not connected to the Internet. However, Jabber does support server to server communication, which means that users on a particular server can have IM friends on other servers. Figure 2 illustrates this.

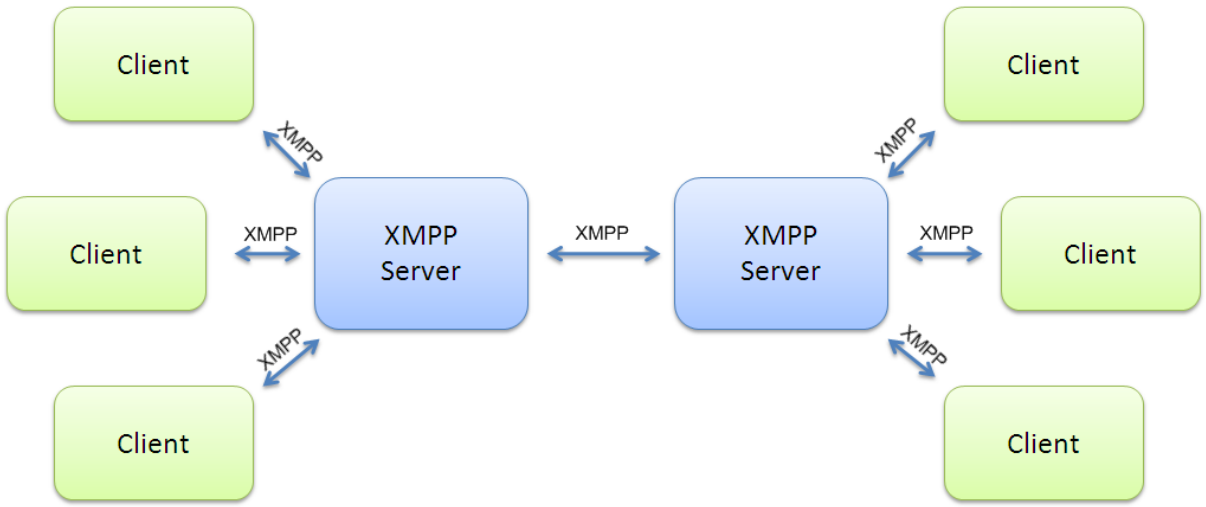

Figure 2: XMPP server to server communication (http://www.isode.com/whitepapers/xmpp.html)

2.2 XEP

XMPP's core protocol and functionality can be enhanced and expanded by extensions. These extensions are called XMPP Extension Protocols (XEP's). They add to the core protocols functionality in some way without making changes to the server/client implementation they are working alongside. This means that the source code of the server/client also can be changed by its author(s) without it affecting the extensions that users might use with it. This thesis will focus on two XEP's made for file transfer. These are XEP-0047 In-band bytestream (XEP-0047) and XEP-0065 Socks 5 bytestream (XEP-0065). The two differ in that XEP-0047 offers in band byte stream (IBB) and XEP-0065 out of band (OOB) byte stream. With XEP-0047, the byte stream will flow through the XMPP server, via the XML stream. With XEP-0065, the byte stream, or file, will go directly from peer to peer, or mediated through a proxy server. We will look further into each of these methods and show some of the XML code that streams between the server and clients.

2.2.1 XEP-0047 in band byte stream

The XEP-0047 specification [11] introduces IBB as a reliable bytestream protocol between two Jabber entities over a Jabber XML stream, as illustrated in Figure 3. The specification/it states that the basic idea is that binary data is encoded as Base64 and transferred over the Jabber network, and that it is likely to be useful for sending small payloads such as binary files. Moreover, the author says that XEP-0047 is mostly intended as a fallback in situations where a SOCKS5 Bytestream [12] is unavailable, and not for byte streams that have a high bandwidth requirement.

The reason it has to be encoded is because of an inherent attribute of binary files that makes it impossible to send raw over the XML stream. This will be explained in more detail as we look at Base64 [13].

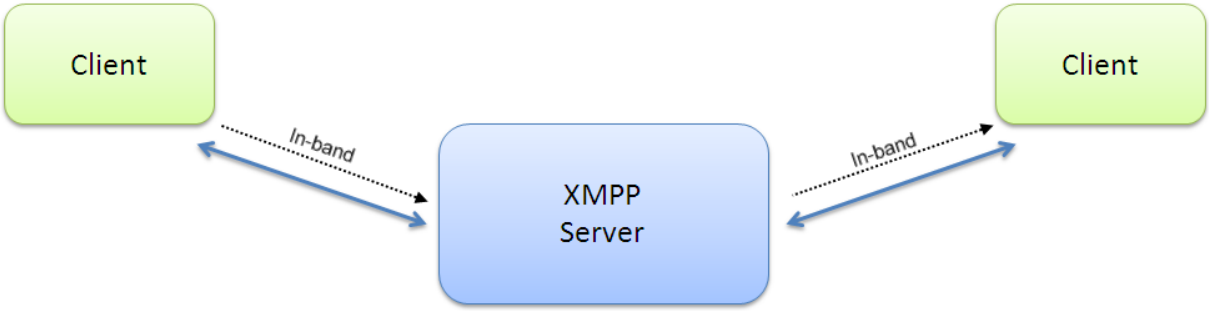

Figure 3: In-band bytestream, from section 2.3.1 in [1]

The term “In-band” means that the data is transported via the XMPP stream, through the server. In contrast, “Out-of-band” means that another connection is established for the actual binary data transfer, that is, outside the stream and XMPP server. “In-band” is not an advantage in terms of performance, since the server already has to deal with every user’s messages, iq and presence data and could be slowed down by the extra load. This is especially true for servers that handle a large amount of requests. However, there is one area where in-band has an advantage over out-of-band and that is where anonymity is concerned. Since both clients only communicate with server, within the XML stream, there is no need to reveal their IP-addresses to each other as would be necessary in a peer-to-peer scenario.

Code listing 1 is XML for creating an in-band bytestream and asks Juliet if she would like to form a connection, using the session id “mySID” to uniquely reference the bytestream. The “block-size” attribute specifies the maximum amount of data (in bytes) that an IBB packet may contain.

```
<iq type='set'
    from='romeo@montague.net/orchard'
    to='juliet@capulet.com/balcony'
    id='inband_1'>
  <open sid='mySID'
        block-size='4096'
        xmlns='http://jabber.org/protocol/ibb'/>
</iq>
```

Code listing 1: Initiaton of interaction, from section 3.1 in [11]

The success response saying that the bytestream is active is shown in Code listing 2.

```
<iq type='result'
    from='juliet@capulet.com/balcony'
    to='romeo@montague.net/orchard'
    id='inband_1'/>
```

Code listing 2: Success response, from section 3.1 in [11]

Data is sent using either <message> or <iq> stanzas. Code listing 3 shows data sent using the message stanza. Data to be sent must not be larger than the "block-size" defined during the in-band bytestream initiation.

```
<message from='romeo@montague.net/orchard'
  to='juliet@capulet.com/balcony' id='msg1'>
  <data xmlns='http://jabber.org/protocol/ibb' sid='mySID' seq='0'>
    qANQR1DBwU4DX7jmYZnncmUQB/9KuKBddzQH+tZ1ZywKK0yHKnq57kWq+RFtQdCJ
    WpdWpR0uQsuJe7+vh3NWn59/gTc5MDlX8dS9p0ovStmNcyLhxVgmqS8ZKhsblVeu
    IpQ0JgavABqibJolc3BKrVtVV1igKiX/N7Pi8RtY1K18toaMDhdEfhBRzO/XB0+P
    AQhYlRjNacGcslkhXqNjK5Va4tuOAPy2n1Q8UUrHbUd0g+xJ9Bm0G0LZXyvCWyKH
    kuNEHFQiLuCY6Iv0myq6iX6tjuHehZlFSh80b5BVV9tNLwNR5Eqz1klxMhoghJOA
  </data>
  <amp xmlns='http://jabber.org/protocol/amp'>
    <rule condition='deliver' value='stored' action='error'/>
    <rule condition='match-resource' value='exact' action='error'/>
  </amp>
</message>
```

Code listing 3: Sending data using message stanza, from section 3.2 in [11] .

The data between the <data> tags is encoded in Base64 as specified in RFC 4648 [13]. According to Morin [14], the problem with binary attachments is that they don't read well when encapsulated in other documents. They need to be encoded, and base 64 is the standard for doing this for SMTP, XMPP and many other internet protocols. Base 64 is a data representation protocol that allows binary data to be encapsulated within another document. The encoding process represents 24-bits groups of input bits as output strings of 4 concatenated 6-bit groups, each of which is translated into a single character in the base 64 alphabet. Each 6-bit group is used as an index into an array of 64 printable characters. The character referenced by the index is placed in the output string [13]. Figure 7 shows the index value with the encoded base 64 character.

Base 64 encoding inflates the data size with about 30 percent, which means that ten megabyte of data turns into thirteen when it is to be transferred in-band between two clients.

Table 2: The Base 64 alphabet, from section 4 in [13] .

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

Base 64 is not necessary with XEP-0065, since it relies on peer-to-peer or proxy mediation. The files are not part of any document or XML stream and are sent as they are. The XML stream negotiates the connection, but the files are transferred directly or through a proxy server.

2.2.2 XEP-0065 Socks 5 bytestream

XMPP is not designed for sending binary data. It is designed for sending relatively small fragments of XML between network entities. For the purpose of file transfer within XMPP a few extension protocols have been created and XEP-0065 Socks 5 bytestream is one [12]. The specification says that XEP-0065 is an XMPP protocol extension for establishing an out-of-band bytestream between any two XMPP users, mainly for the purpose of file transfer. The bytestream can be either direct (peer-to-peer) or mediated through a special-purpose proxy server. The typical transport protocol used is TCP, although UDP may optionally be supported as well.

To accurately communicate what is going on in these two scenarios a list of terms is defined as shown in table 3.

Term	Description
Initiator	A Jabber Entity that wishes to establish a bytestream with another Entity
Target	The Entity with which the Initiator is attempting to establish a bytestream
Proxy	A Jabber entity which is not NAT/Firewalled and is willing to be a middleman for the bytestream between the Initiator and the Target
StreamHost	The system that the Target connects to and that is "hosting" the bytestream -- may be either the Initiator or a Proxy
StreamID	A relatively unique Stream ID for this connection; this is generated by the Initiator for tracking purposes and MUST be less than 128 characters in length

Table 3: Terms used in description of XEP-0065 connection scenarios, from section 2 in [12].

The direct connection model would look like Figure 3. The model of the mediated connection is shown in Figure 4.

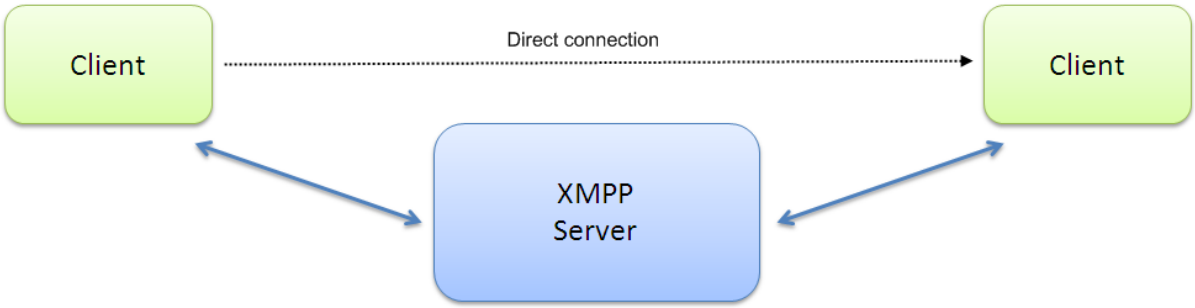

Figure 4: Direct connection, from section 2.3.2 in [1].

Direct connection is according to [12] the simpler case of the two. The specification details the process for establishing this kind of bytestream:

1. *Initiator sends IQ-set to Target specifying the full JID and network address of StreamHost/Initiator as well as the StreamID (SID) of the proposed bytestream.*
2. *Target opens a TCP socket to the specified network address.*
3. *Target requests connection via SOCKS5, with the DST.ADDR and DST.PORT parameters set to the values defined below.*
4. *StreamHost/Initiator sends acknowledgement of successful connection to Target via SOCKS5.*
5. *Target sends IQ-result to Initiator, preserving the 'id' of the initial IQ-set.*
6. *StreamHost/Initiator activates the bytestream.*
7. *Initiator and Target may begin using the bytestream.*

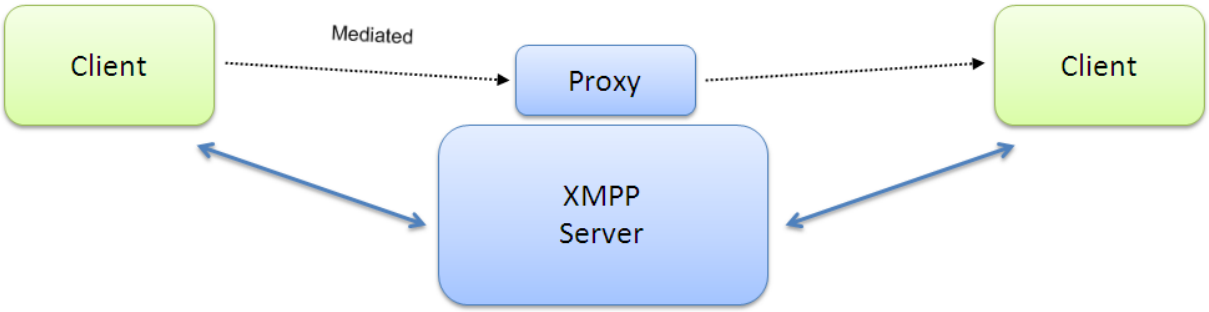

Figure 5: Mediated connection, from section 2.3.2 in [1].

Mediated connection, as shown in Figure 5, involves a proxy that acts as the StreamHost, as opposed to direct connection where the Initiator has that role, as. That means that both Initiator and Target must negotiate a connection with the StreamHost, which again means that they need to know the network address of the proxy, or StreamHost. The process of establishing a mediated bytestream is taken from section 3.2 in [12].

1. *Optionally, Initiator discovers the network address of StreamHost in-band.*
2. *Initiator sends IQ-set to Target specifying the full JID and network address of StreamHost as well as the StreamID (SID) of the proposed bytestream.*
3. *Target opens a TCP socket to the selected StreamHost.*
4. *Target establishes connection via SOCKS5, with the DST.ADDR and DST.PORT parameters set to the values defined below.*
5. *StreamHost sends acknowledgement of successful connection to Target via SOCKS5.*
6. *Target sends IQ-result to Initiator, preserving the 'id' of the initial IQ-set.*
7. *Initiator opens a TCP socket at the StreamHost.*
8. *Initiator establishes connection via SOCKS5, with the DST.ADDR and DST.PORT parameters set to the values defined below.*
9. *StreamHost sends acknowledgement of successful connection to Initiator via SOCKS5.*
10. *Initiator sends IQ-set to StreamHost requesting that StreamHost activate the bytestream associated with the StreamID.*
11. *StreamHost activates the bytestream. (Data is now relayed between the two SOCKS5 connections by the proxy.)*
12. *StreamHost sends IQ-result to Initiator acknowledging that the bytestream has been activated (or specifying an error).*
13. *Initiator and Target may begin using the bytestream.*

The XML code below is taken from the XEP-0065 RFC and is included to give deeper insight into the protocol extension.

The Initiator may want to know if the Target supports the bytestream protocol. It may do so using Service discovery [15] as follows and shown in Code listing 4-7:

```
<iq type='get'
    from='initiator@example.com/foo'
    to='target@example.org/bar'
    id='hello'>
  <query xmlns='http://jabber.org/protocol/disco#info'/>
</iq>
```

Code listing 4: Initiator sends service discovery request to target, from section 4.1 in [12].

If the Target supports it, it will answer like shown in Code listing 5.

```
<iq type='result'
    from='target@example.org/bar'
    to='initiator@example.com/foo'
    id='hello'>
  <query xmlns='http://jabber.org/protocol/disco#info'>
    <identity
        category='proxy'
        type='bytestreams'
        name='SOCKS5 Bytestreams Service'/>
    ...
    <feature var='http://jabber.org/protocol/bytestreams'/>
    ...
  </query>
</iq>
```

Code listing 5: Target replies to service discovery request, from section 4.1 in [12].

Initiator locates proxy using service discovery before initiating a bytestream:

```
<iq type='get'
    from='initiator@example.com/foo'
    to='example.com'
    id='server_items'>
  <query xmlns='http://jabber.org/protocol/disco#items'/>
</iq>
```

Code listing 6: Initiator sends service discovery request to server, from section 4.2 in [12]

Server returns known JIDs.

```
<iq type='result'
    from='example.com'
    to='initiator@example.com/foo'
    id='server_items'>
  <query xmlns='http://jabber.org/protocol/disco#items'>
    ...
    <item jid='streamhostproxy.example.net' name='Bytestreams Proxy'/>
    ...
  </query>
</iq>
```

Code listing 7: Server replies to service discovery request, from section 4.2 in [12]

2.3 Health Network

The following is a quote from section 2.1.3 in [1]:

> *A Health network (HN) can be seen as an Internet Service Provider (ISP) for the health regions in Norway, where its main purpose is to provide services for communication between primary healthcare and the specialist healthcare providers. One of the most important forms of communication is the transfer of medical information to and from the hospitals. Examples of data in this communication are discharge letters, lab results, referrals and radiology results. This communication is performed today using the well known protocols POP and SMTP. Due to the sensitivity of data communicated within the framework of the HN, there are strict safety measures established for securing the reliability of the services and the protection of the data communicated within the network [16]. This security consists of several different items, where firewalls, traffic filtering, logging and encryption are some of them. The network is also divided into different zones, based on the level of trust between the communication partners.*

All five health regions in Norway established their own regional health networks during the second half of 1990. The regions had no inter-regional coordination, so technology, service, ambition and organization differed. In 2003 the National Health Network project was initiated by the Norwegian government to establish a central infrastructure that would connect the health networks. In 2004, the six network conglomerate, including equipment, personnel and contracts, was transferred to Norsk Helsenett AS, which at the time was a newly founded company [16].

Most hospitals and other units in the specialist care service still have their connection to Norsk Helsenett through the old regional health network structure. In 2006 the central infrastructure was moved to the IP-VPN service Nordic Connect, which is supplied by Telenor. The intention is to phase out the old regional networks, and have the hospitals connect directly to this [16].

During the implementation phase of a new service into the operating environment of a health region there are several issues to look into. One is the level of security enabled on the traffic flow within the new system, what gateways will be passed through, what firewalls have to allow traffic and what systems will be exposed to the new service. One could grade the intrusion into existing security systems into three stages for classifying port access through firewalls.

The following is a quote from 2.1.3 in [1]:

- *0 – No openings, would be the very best and of course keep the systems optimal safe*
- *1 – One opening, giving new implementations access but at the same time keeping security to a satisfying level. Would of course need risk and security analysis.*
- *Many – More than one opened port for access would demand comprehensive risk and security analysis in addition to a much higher demand for resources in the operating department.*

According to [1], an application that would support several services running through the same firewall accessed port could reduce the cost of resources needed by the operational department. He also states that it could decrease the need for preparatory work for the new service to be accepted as a part of the risk and security level that is maintained.

2.4 Openfire

Openfire [3], formerly known as Wildfire, is an open source, real time collaboration (RTC) XMPP server created by Jive Software [17]. The latest version is at the time of writing 3.6.4 and was released May 1, 2009. There are other open source XMPP server implementations available, like Tigase and ejabberd, but we choose to work with Openfire for reasons previously explained.

At the time we starting working with the Openfire server source code, it had reached version 3.6.0, and we have kept that version throughout to avoid overwriting changes, comments and debugging code made by us. An issue with this approach is that the changes and improvements made by the developers might coincide with our changes and improvements, thereby rendering our work less relevant or completely irrelevant, but this did not seem too likely.

Openfire is freely supported through its community forum, but commercial support is also available from Jive Software.

Openfire supports the following features [18]:

- Web-based administration panel
- Plug-in interface
- Customizable
- SSL/TLS support
- User-friendly web interface and guided installation
- Database connectivity (i.e. embedded Apache Derby or other DBMS with JDBC 3 driver) for storing messages and user details
- LDAP connectivity
- Platform independent, pure Java
- Full integration with Spark Jabber client

The Openfire source code is governed by the GNU Public License (GPL) [19].

In-band bytestream file transfer is inherently slower than its counterpart out of band file transfer. In Openfire both XEP-0047 and XEP-0065 are implemented alongside each other as it is described in.

Openfire has implemented the XEP-0096 [20] SI File Transfer extension. This is an extension that attempts to overcome some of the shortcomings of out of band file transfer, like the fact that it does not work when one of the parties is behind a firewall and it's not reliable. Simply put, the extension makes sure that whenever it is not possible to use out of band file transfer, Openfire will fall back on

in-band file transfer instead. Since we will deal with in-band bytestream file transfers exclusively, we have disabled Openfire's ability to use out of band file transfers.

2.5 Spark

Spark [21] is the XMPP client that for the most part has been used in this experiment. It is made by Jive Software and has reached version number 2.5.8. The latest version was released on November 14th, 2007.

There is a wide range of XMPP clients available, and of varying quality. Some are one-man projects, some are collaborations and then others are made by corporations, like our choice. We have not gone to great lengths to find out which client would serve us best, as Spark was the first one we tried, downloaded from the same site as Openfire, and it has worked well for our purpose, which was to test file transfer between XMPP clients. In an attempt to run two clients alongside each other on the same installation, something Spark would not allow us to do, we tried a few others, like Exodus [22]. Exodus met this requirement but seemed to violate the specification for in band file transfer, as tests revealed that the file data were encoded in something other than base64. We tried to have the Spark client send a file to Exodus client and it did not work when we had made sure in the server settings that only in band transfers were allowed. We turned back to Spark and solved the problem of running more than one client by installing a virtual machine and ran the second Spark client from that. Now we had a client-server-client communication setup.

3 Requirements and specifications

There is one main requirement for this thesis, and that is to improve in-band bytestream file transfer performance in the Openfire XMPP server. There is a significant performance decrease in switching from out-of-band to in-band bytestream. This drop in performance will most likely vary from server implementation to implementation, as there inevitably will be some differences in how the XEP-0047 is implemented in the various servers. Our goal is to narrow the gap between Openfire's out-of-band and in-band bytestream efficiency, to make the latter a more viable alternative to the more common way of transferring files in the health network environment, which is currently e-mail and e-mail attachments.

4 Design

In this thesis we have used the Openfire XMPP server and the Spark XMPP clients for reasons previously explained. This chapter will detail the design of these components in regard to file transfer. Figure 6 shows an overview of a client-to-server-to-client setup, using in-band file transfer. The arrows indicate the bytestream that would flow to and from the clients, through the Openfire server.

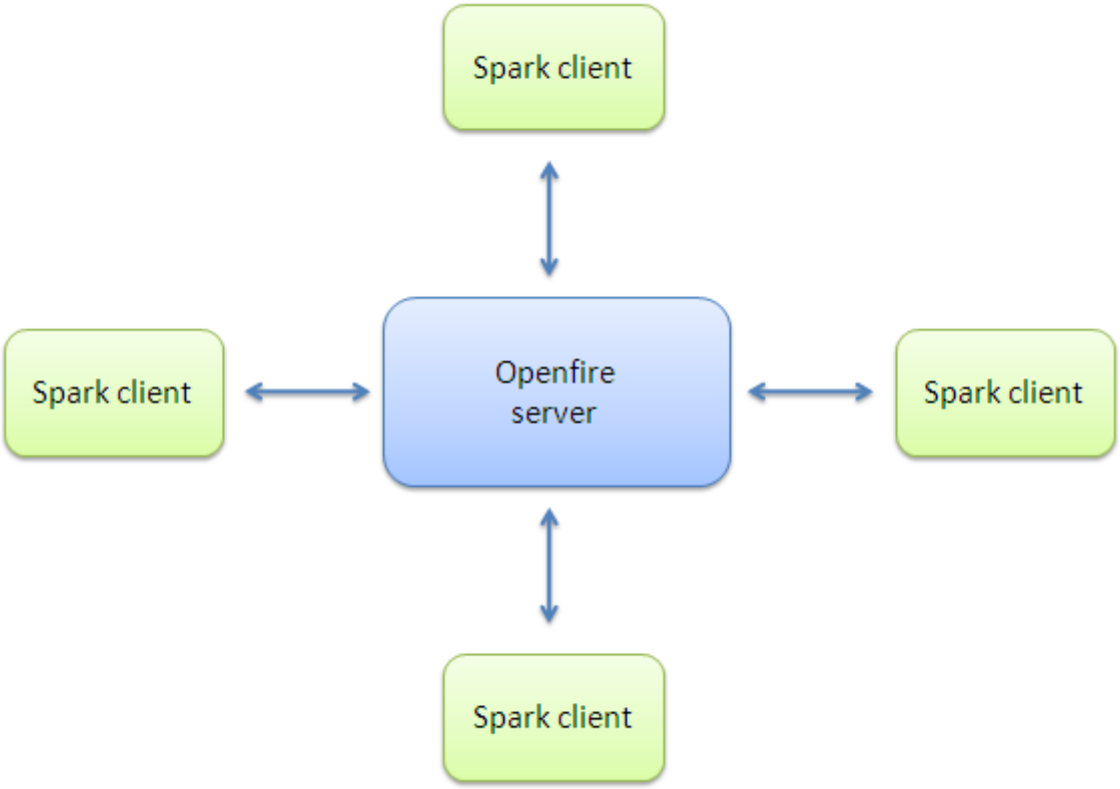

Figure 6: Spark clients and Openfire server, arrows indicate file transfer stream.

Figure 7 shows a closer look at the Spark client. It emphasizes the base 64 encoding and decoding process that takes place in the client during an in-band file transfer, an encoding process which increases the total file size with about one third. A string like: “This string of bytes is about to be encoded in base 64” is 54 bytes long. Encoded in base 64 it looks like: “VGhpcyBzdHJpbmcgb2YgYnl0ZXMgaXMgYWJvdXQgdG8gYmUgZW5jb2RlZCBpbiBiYXNlIDY0” and is 72 bytes long, which is exactly one third of increase.

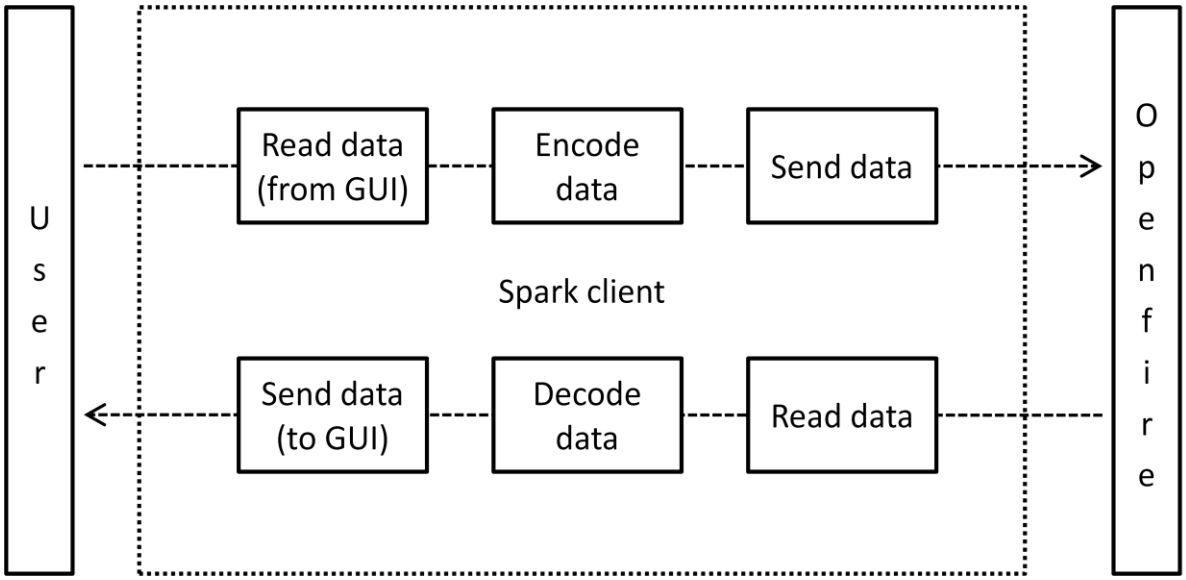

Figure 7: Spark client in-band file transfer operation.

Figure 8 shows the Openfire XMPP server sending and receiving to and from the spark clients. Notice that the data is processed on its way through the server. In fact, we observed that Openfire's interceptPacket method, in *"/openfire/src/java/org/jivesoftware/openfire/filetransfer/ DefaultFileTransferManager.java*" found four packages for every package sent from a client (a total of four, including the one from the sending client). So a message package sent containing encoded file data would appear four times when showing the output of these packages. Closer inspection revealed that two of these packages were linked to the sender's session and the other two to the receiver's session. Every package had a Boolean value for whether it was an incoming package and whether it was processed or not.

Another thing to keep in mind is that Openfire is limited by its assigned virtual memory as to how large files it can handle. If the server is set to run with 1024 megabytes of memory, it will crash in an attempt to transfer larger files, see section 5.1 in [1].

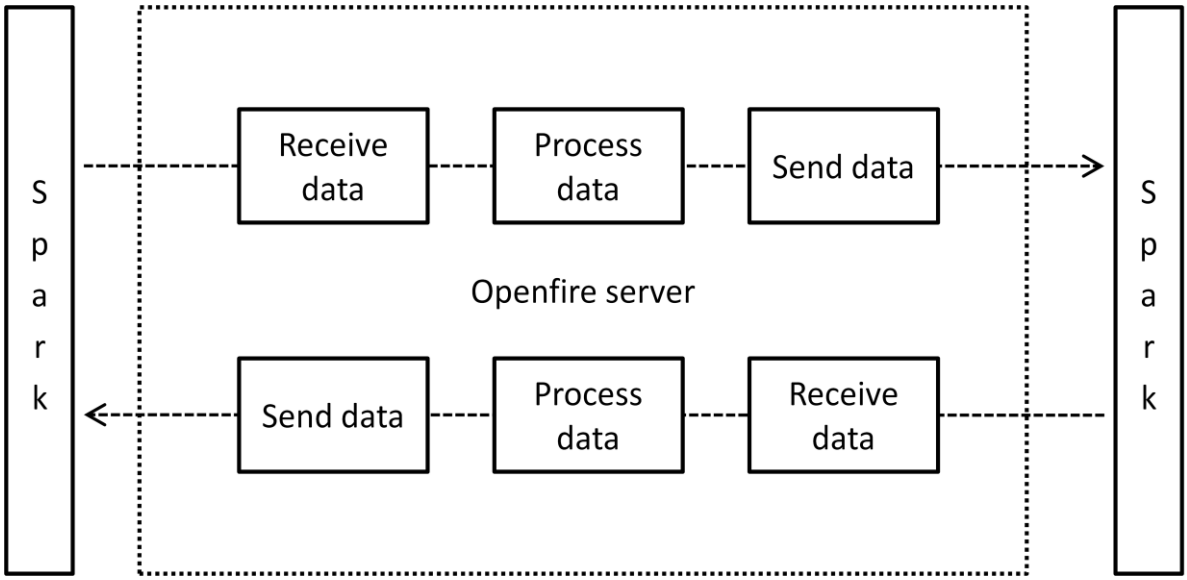

Figure 8: Openfire client to server to client in-band file transfer

5 Implementation

This chapter aims to give you some insight into our experience with Openfire.

Openfire is a large and complex system. The compressed and zipped source code weighs in at an, from a developers point of view, intimidating 51 megabyte. That amount might not seem like much in terms of sheer byte size, but it is a lot of code to take in and understand, and obviously, one can't be expected to gain a complete understanding of such a massive system in the relative short amount of time that is available. As a consequence, we focus on what would seem like the most relevant parts of the system, those that deal with file transfer, see Figure 9.

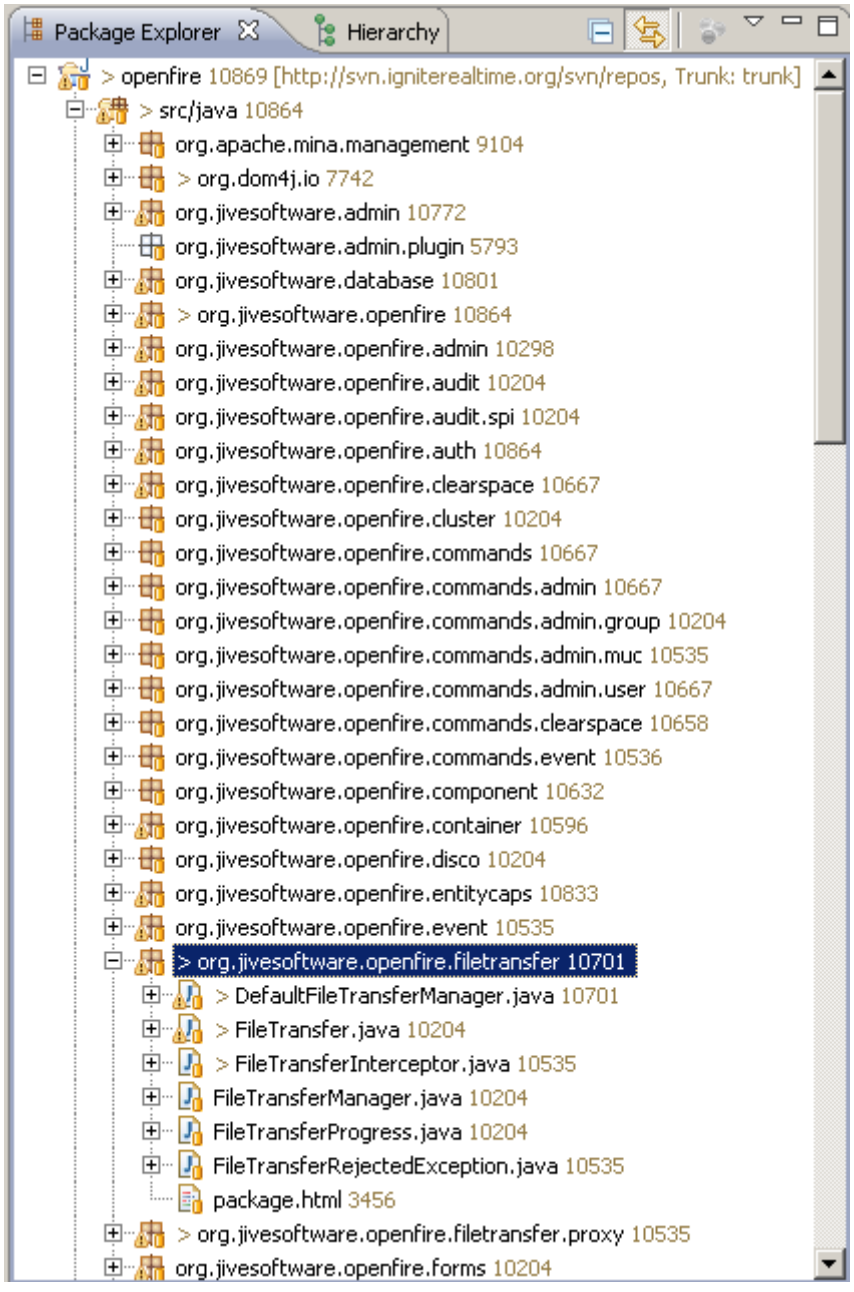

Figure 9: Some of Openfire's packages, with one file transfer packages highlighted

What I found to be challenging in researching the Openfire code, was how strikingly modulated the code is. One simple method can include several other method calls and quite a few classes and libraries. That made it hard to see what is really going on, since a lot of code is elsewhere in the aforementioned methods, classes and libraries. This modularity is illustrated in Figure 6, which also shows what we found to be a helpful method of gathering the threads to attain a clearer and more complete picture of what is going on at a certain place in the code.

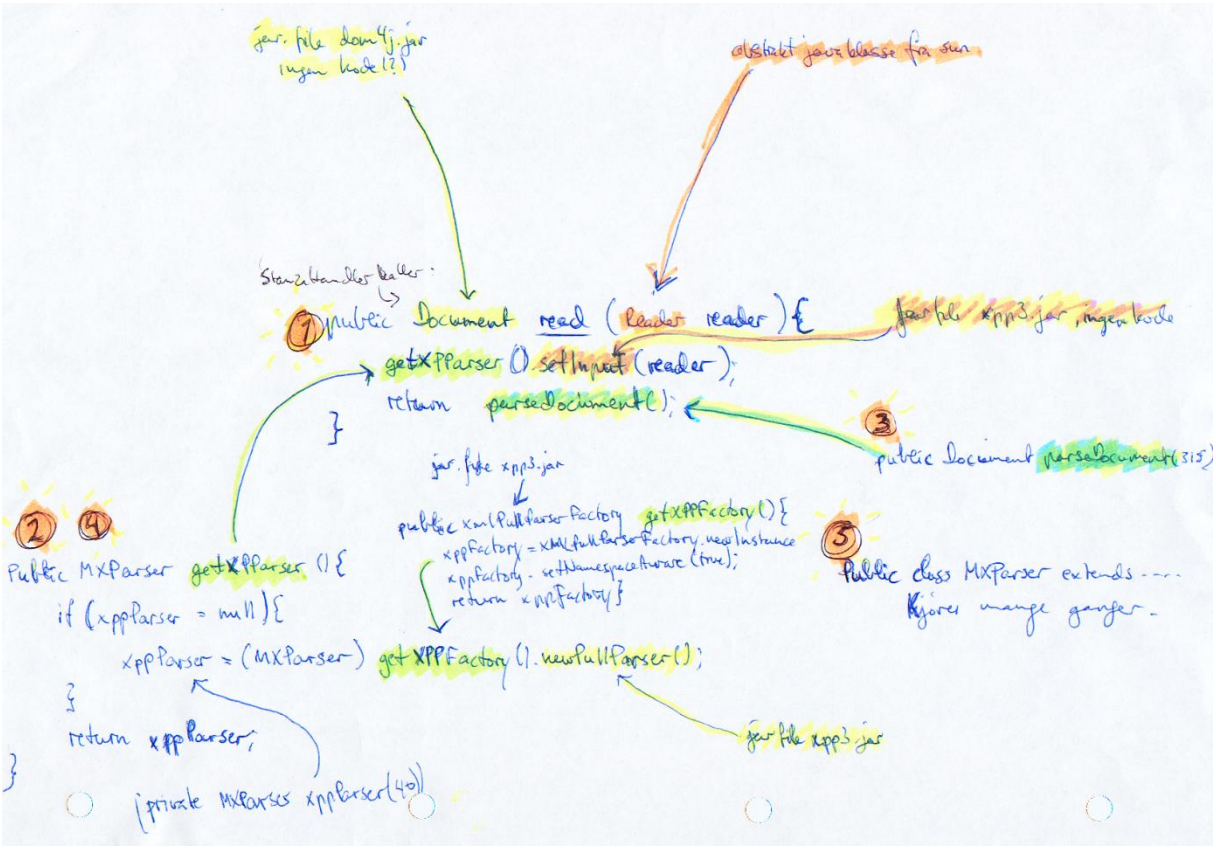

Figure 10: Overview of interconnected classes, methods and libraries, numbers 1-5 shows in which sequence the code is running

We never found a way to optimize the file transfer performance through the source code. One problem was that too much time got spent over in what proved to be the wrong package, the *org.jivesoftware.openfire.filetransfer* package, and only late in the process did we realize that this code didn't really do anything significant to the in-band transfers. It seemed like a natural place to search for leads. However, that code seems to primarily deal with metainformation to the transfers, and commenting it out what previously seemed like important code made no difference to the outcome of a file transfer. It became clear to us that the XMPP file transfer is handled in mainly the same way as the rest of the XMPP stream; the byte stream is encoded in base64 and sent together with the rest of the iq and message stanzas, which led us to the code dealing with general message handling. Eventually we found a place in the code that made a difference, but only in terms of whether text was transferred or whether files transfers were completed.

The method marked 1 from Figure 10, *public Document read(Reader reader),* is found in the class called XMPPPacketReader, in the package *org.dom4j.io* in the Openfire hierarchy. This method, as partially shown in Code Listing 8 returns a method called *public Document parseDocument()*. It seems to handle the XMPP stanzas, or packets. Note the switch statement, testing for the type of XmlPullParser.

```
public Document parseDocument() throws DocumentException, IOException
{
    DocumentFactory df = getDocumentFactory();
    Document document = df.createDocument();
    Element parent = null;
    XmlPullParser pp = getXPPParser();

    int count = 0;

    while (true){
        int type = -1;
        type = pp.nextToken();

        switch (type){
            case XmlPullParser.PROCESSING_INSTRUCTION:
```

Code listing 8: The first part of the parseDocument() method

Debugging shows that the most commonly used types seems to be XmlPullParser.START_TAG, END_TAG AND TEXT, at least for normal messaging and file transfers. The XmlPullParser.TEXT is shown in Code listing 9. Both messages and the base64 encoded file content are in the text variable.

```
            case XmlPullParser.TEXT:
            {
                String text = pp.getText();
                if (parent != null){
                    parent.addText(text);
                }
                else{
                    if (text.trim().length() > 0){
                        throw new DocumentException("C
                    }
                }
                break;
            }
```

Code listing 9: case XmlPullParser.TEXT

From here on there wasn't much more that we could do. We could conclude that commenting out the line parent.addText(text) prevents messages and files from being sent to the client, but other

than that, there wasn’t any apparent room for improvement. At this level we bumped into standard Java libraries (jar files) just about everywhere we looked, which is a problem when you looking for things you can change.

6 Testing and results

Testing did not work out as originally planned. We had purposed to have a testing environment set up within the Norwegian Health Network, to measure the performance of our improved version of Openfire's in-band file transfer against the normal one, but this was evidently not necessary seeing there was no new software solution to test. We could of course set up a cluster with the Openfire Enterprise server and test performance, but that would be a lot of work and effort to prove something already quite evident, that a cluster performs better than a single server, at least when there are a lot of users and traffic involved.

7 Discussion

In terms of performance, in-band file transfer's main problem is that file traffic has to be routed through the server. That is why its specification states that it should only be used as a fallback solution to out-of-band file transfers, and that is also the way Openfire uses it by default. Our goal was to see if it was possible to improve in-band file transfer performance by tweaking and changing the source code of Openfire. This proved problematic. One reason is the massive amount of source code - this system is huge, and we found it hard to completely understand and comprehend. Another is the modularity of the code; everything is interconnected, and finding out what connects to what was quite a challenge. Thirdly, when we eventually found our way through parts of the code jungle, we often got to a dead end, represented by the use of standard java libraries, which we obviously could not do much to. That leaves us with only a few more options for improving in-band file transfer performance; upgrading the server machine – more memory, faster processor and so on, removing bottlenecks like a slow network connection or both of the above combined with clustering; many interconnected servers using dynamic load-balancing. This feature has been available in Openfire Enterprise since version 3.4.0, but only commercially, hence the Enterprise name. A clustered Openfire Enterprise would probably be the way to go if XMPP were to replace or complement e-mail as the primary electronic communication protocol in the Norwegian Health Network. Openfire Enterprise is commercial and supported by Jive Software. Whether users would convert from e-mail to XMPP is an entirely different discussion.

The base 64 encoding/decoding used for in-band file transfer is an interesting subject, since a there is a significant extra amount of data produced from this process. One possible solution that comes to mind about this problem is file compression prior to encoding and sending to server, to compensate for the increased file size. The XMPP client would first compress the file, and then encode it with base64. The receiving client would decode the file content, and then decompress it. This would put a heavier load on the clients and infuse a time delay, but would free up resources on the server in the cases where there actually was something to gain from compressing. In some cases, as with certain picture formats, there is not a whole lot to gain in compressing, as they are already as compressed as they can be.

8 Conclusion

In hindsight we may have to conclude that the assignment for this thesis was a bit too ambitious. However, it has to a certain extent proved that there is no easily spotted way to enhance in-band file transfer performance in the Openfire server source code. I say that hesitantly, as I am no expert on either Java or Openfire, but from what I have seen so far, I am not encouraged to keep looking.

References

1. Andreassen, K., *The Reliability of XMPP for file transfer*, in *Department of Computer Science*. 2008, University in Tromsø: Tromsø. p. 80.
2. Denning, P.J., et al., *Computing as a discipline.* Communications of the ACM, 1989. **32**(1): p. 9-23.
3. *Ignite Realtime Openfire Server*. [cited 2009 09/01/09]; Available from: http://www.igniterealtime.org/projects/openfire/index.jsp.
4. *tigase.org Open Source and Free (GPLv3) Jabber-XMPP environment.* [cited 2009 09/01/09]; Tigase.org is the official website of the Tigase open source XMPP/Jabber projects:]. Available from: http://www.tigase.org/.
5. *ejabberd - the Erlang Jabber/XMPP daemon*. [cited 2009 23/09]; Available from: http://www.ejabberd.im/.
6. *Eclipse - Java IDE*. p. Java IDE.
7. *Slashdot - Open Real Time Messaging System* 1999. **2008**(29/11/08).
8. Saint-Andre, P., *Streaming XML with Jabber/XMPP.* IEEE Internet Computing, 2005. **9**(5): p. 82-9.
9. xmpp.org. *XMPP RFCs*. [cited 2008 01/12]; Available from: http://xmpp.org/rfcs/.
10. Saint-Andre, P. *Extensible Messaging and Presence Protocol (XMPP): Core*. 2004; Available from: http://xmpp.org/rfcs/rfc3920.html.
11. Karneges, J. *XEP-0047: In-Band Bytestream (IBB)*. Available from: http://xmpp.org/extensions/xep-0047.html.
12. Smith, D.M., Mattew Saint-Andre, Peter. *XEP-0065: SOCKS5 Bytestreams*. Available from: http://xmpp.org/extensions/xep-0065.html.
13. Josefsson, S., *RFC4648: The Base16, Base32, and Base64 Data Encodings*, ed. S. Josefsson. 2003: RFC Editor.
14. Morin, R.C., *How to Base64.*
15. Hildebrand, J., et al. *XEP-0030: Service Discovery*. Available from: http://xmpp.org/extensions/xep-0030.html.
16. Baardsgaard, A., *The Scandinavian health network : connecting the Scandinavian countries' health networks*. 2007, [Tromsø]: A. Baardsgaard. 77 s.
17. *Jive Software*. [Webpage] [cited 2009 09/01/09]; Available from: http://www.jivesoftware.com/.
18. *About Openfire on Wikipedia*. [cited 2009 29/08]; Available from: http://en.wikipedia.org/wiki/Openfire.
19. *GNU General Public License*. [cited 2009 29/08]; Available from: http://www.gnu.org/licenses/gpl.html.
20. Muldowney, T. *XEP-0096: SI File Transfer*. [cited 2009 01/09].
21. *Spark IM Client*, Jive Software. p. Spark is a full-features instant messaging (IM) and groupchat client that uses the XMPP protocol. .
22. Millard, P., *Exodus XMPP IM Client*. p. Exodus is a free software instant messaging client developed by Peter Millard and written in Borland Delphi that can connect to XMPP servers and exchange messages with other XMPP users. Currently, binaries are only available for Microsoft Windows. Exodus is licensed under the GNU GPL.